Bulletin of Aichi Univ. of Education, 65（Natural Sciences）, pp. 25 − 28, March, 2016

Common Lisp言語処理系によるBroadwellの任意精度整数演算命令の評価

安本 太一

情報教育講座

Evaluation of Broadwell Instructions Supporting Large Interger Arithmetic on a Common Lisp System

Taichi YASUMOTO

Department of Information Sciences, Aichi University of Education, Kariya 448-8542, Japan

1 はじめに

インテル社の第5世代CoreプロセッサBroadwellマイクロアーキテクチャに実装されている任意精度の整数演算のための命令[1, 2]の評価を，Common Lisp[3]言語処理系であるKCL（Kyoto Common Lisp）[4, 5]のbignum（無限長整数）の掛け算を用いて行ったので，その結果を報告する．なお，本論文では，任意精度整数は一般的な用語，bignumはCommon Lispの無限長整数を示すものとして扱う．

任意精度の整数演算をとりあげるとき，基本データ型である整数の配列(例えば32ビットや64ビットの整数の配列）で，基本データ型の整数よりも長い任意精度の整数を表現することが多い［1, 2］．基本データ型である整数の演算を何度も適用して，任意精度の整数の演算を行うのである．一方，KCLのbignumは，配列でなく，リスト形式で，C言語の基本データ型の整数を連結した表現で実現されている．

配列形式の任意精度整数において，インテル社が提案して実装した任意精度の整数演算のための命令が速度向上に寄与するのは，掛け算に必要な命令数が減っていることがインテル社のドキュメントおいて示されているので，明らかである［1］．

そこで，今回は，これらの命令を，実際のLisp言語処理系において，リスト形式で表現されているbignumにおける掛け算で用いた場合は，どれくらいの速度向上が得られるのか調べてみた．

2 Broadwellにおける任意精度整数演算のための命令

2.1 mulx, adcx, adox命令

Broadwellには，任意精度整数演算をサポートするための命令として，mulx，adcx，adoxが用意されている．

mulxは，64ビット（あるいは32ビット）の符号なし整数どうしsrc1，src2の掛け算を行う機械語命令mulの拡張版である．アセンブラ表記は次の通りであり，64ビットの場合はrdxを，32ビットの場合はedxをsrc2として使う．

```
mulx dest_hi，dest_lo，src1
```

その動作は

```
dest_hi:dest_lo = src1 * r/edx
```

のように，ソースオペランドsrc1とsrc2の積を計算し，その結果を，2つのレジスタを連結したディスティネーションオペランドであるdest_hi:dest_loに格納する．フラグレジスタは変更されない．実質4オペランドの命令なので，ソースオペランドの値を破壊しないことが可能である，

adcxとadoxは，adc（キャリー付き加算命令）の拡張版であり，アセンブラ表記は次の通りである．

```
adcx dest/src1，src2
adox dest/src1，src2
```

これらの2つの命令は，src1とsrc2のキャリー付き加算を行い，その結果をdestに格納する．adcxとadoxの違いは，CFフラグとOFフラグの扱いである．adcxは，CFフラグを用いてキャリーインしてキャリーアウトし，OFフラグを変更しない．adoxは，OFフラグを用いてキャリーインしてキャリーアウトし，CFフラグを変更しない．adcxとadoxを使い分けることによって，2つのキャリーフラグのチェーン（流れ）を作ることができる．

例えば，mulx，adcx，adoxを使うと，64ビッ

トの整数を8個連結して表現した512ビットの整数$A_7A_6...A_1A_0$，$B_7B_6...B_1B_0$の積を，効率よく計算できる．筆算のように，$A_7A_6...A_1A_0 \times B_i$を繰り返して行うとき，$A_7A_6...A_1A_0 \times B_{i-1}$の結果と加算していくことになるが，$A_j \times B_i$は2語になるので，“フラグを変更しない掛け算命令”と“キャリーの流れを2つ実現できる加算命令”があると，命令数が少なくて済むからである［1］．

なお，今後は，32ビットについては言及せず，64ビットにのみに言及する．

2.2 高級言語からの利用

mulx，adcx，adox命令は，次に示すように，C言語（C++言語を含む）から次のような関数呼び出しの形で利用可能であると，ICC（Intel C++ Compiler）を発売しているインテル社のドキュメントに記述されている［1］．gccの場合は，_umul128ではなく_mulx_u64である．関数呼び出しの形式ではあるが，gcc-4.9コンパイラにおいて，call命令を伴わなずに，アセンブラ命令（機械語命令）に展開されることを確認している．

```
/* mulxに対応．返り値がlo_dest */
unsigned __int64
umul128(unsigned __int64 src1,
        unsigned __int64 src2,
        unsigned __int64 *hi_dest);

/* adcxとadoxに対応，返り値がc_out */
unsigned char
_addcarryx_u64(unsigned char c_in,
               unsigned __int64 src1,
               unsigned __int64 src2,
               unsigned __int64 *sum_out);
```

キャリーのためのC言語の変数（c_in, c_outに対応する変数）の名前を使い分けて，_addcarryx_u64を使えば，コンパイラの方で，adcxとadoxを使い分けるコード生成をすると，インテル社のドキュメントには記述されている［6］．ICCの場合は，そうなるのかもしれないが，gcc-4.9ではそのようにはならず，C言語の変数を用いてキャリーフラブを陽に設定するコードが生成されていた．

AVXのSIMD命令に対応するC言語の関数が用意されるといったことはあったが，上記のように逐次実行の機械語命令に対応するC言語の関数が用意されることは珍しい．高級言語から特定の機械語命令を明示的に指定できるので，キャスト演算子を駆使するなどプログラムの書き方を工夫して特定の機械語命令が生成されるようコンパイラに期待するいう，不安定でやりにくい方法をとる必要がない．

3 KCLのbignum

KCLのbignumは，図1に示すbignumセル（無限長整数セル）の連結によって実現されている．この図で示すのは64ビット版のKCLにおける表現であり［7, 8］，本論文では64ビット版のKCLに64ビット版のmulxとadcxを適用することのみを扱う．

固定長の任意精度整数のように64ビットの整数領域を連続してメモリ領域に配置した表現ではなく，連続した領域が確保できなくても，メモリの許す限り無限長の整数を表現できるよう，このような構造になっている．計算の過程で，変数から参照されなくなったbignumが使用していたbignumセルの領域は，ごみ集めで回収され，再利用可能なメモリ領域として利用される．

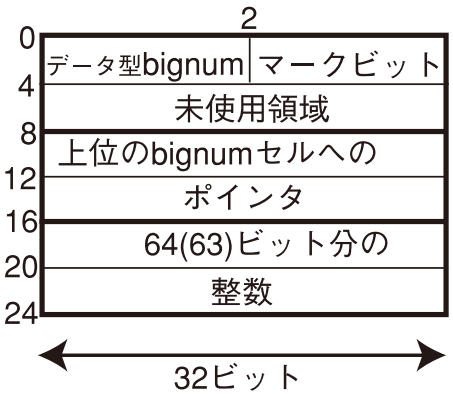

図1：64ビット環境下のbignumセルの構造

bignumを構成する整数を表現するためのフィールドとして64ビット確保しているが，実際に整数を表現しているのは下位63ビットである．上位1ビットは一時的な利用場所で，例えば，下位63ビットに整数を加算した直後のキャリー検出に使用し，キャリー検出に対応した処理を行った後はすぐ0にされる．

bignumの掛け算を実現するための内部ルーチンとして，extended_mul(long int d, long int q, long int r, long int *hp, long int *lp)を使用している．$d \times q + r$を計算し，その結果を，hpとlpが指す場所に格納する．d, q, rは，64ビットの非負整数である．lpが指す場所には，結果の下位63ビットを格納し，最上位ビットは0である．hpが指す場所には，結果の残りのビットを格納し，最上位ビットは0である．

extended_mulは，当初は実行効率を重視してインラインアセンブラで記述されていたが，KCLの移植の容易性を重視してC言語バージョンが用意されている．KCLの配布ソースには，アセンブラバージョンは32ビットプロセッサ向けのみが含まれているので，今回，64ビット版のmulxとadcxの使用を試みることは意義がある．大雑把にいうと64ビット版のC言語のバージョンは，図2に示すように，64ビットの掛け算

を，被乗数dと乗数qを32ビットに分けて計算するようになっていた．オーバーフローを防ぐためである．本図において，rの加算，積の結果のlpやhpへの63ビットずつの配置は省略されている．

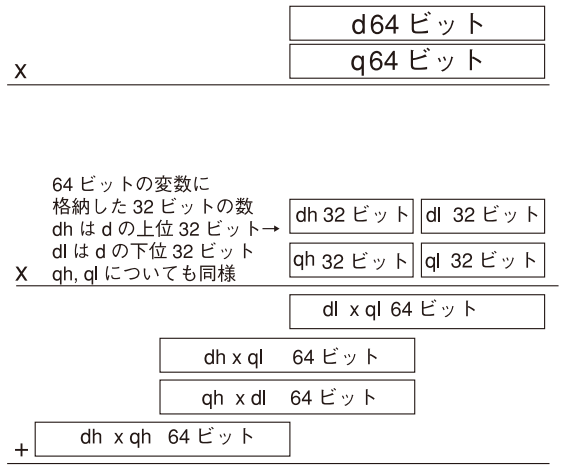

図2：オリジナルのC言語版extended_mulの概要

```
extended_mul(d, q, r, hp, lp)
long int d, q, r;
long int *hp, *lp;
{
  unsigned long long hz, lz;
  unsigned char carry;

  carry = 0;
  lz = _mulx_u64(d, q, &hz);
  carry = _addcarryx_u64(carry, lz, r, &lz);
  _addcarryx_u64(carry, hz, 0, &hz);

  hz = (hz << 1) | (lz >> 63);
  lz = lz & 0x7fffffffffffffff;

  *hp = hz;
  *lp = lz;
}
```

図3：任意精度整数演算のための命令を利用したC言語版extended_mul

4　任意精度整数演算をサポートする命令mulx, adcxの利用

先述の_mulx_u64と_addcarryx_u64を用いて書き換えた，C言語バージョンのextended_mulを図3に示す．gcc-4.9ののヘッダにあわせて，unsigned_int64ではなく，unsigned long longを使用しているが，64ビット環境下では，unsigned long longはlong intとともに，64ビットである．

積$d \times q$は4つに分けることなく_mulx_u64で一気に行う．和$+r$は_addcarryx_u64を用いて行う．キャリーは，C言語の変数carryで明示的に受け渡しする．その結果，extended_mulの（ブロックの内側の）行数は，オリジナルのC言語版と比べて40%程度に減少した．

5　評価実験

mulx, adcxの使用の有無によって，階乗のプログラムの実行時間がどのように変化するか調べた．また，extended_mulを（定義しているKCLのソースファイルearith.cを）gccでコンパイルするときの最適化オプション（-O）の有無によって，階乗のプログラムの実行時間がどのように変わるかも調べた．

実行環境は，アップル社製MacBook Pro（Retina, 13-inch, Early 2015）で，搭載しているCPUはIntel Core i7-5557U 3.1GHzである．OSはOS X 10.10.5，gccのバージョンは4.9である．実験に用いた階乗のプログラムは（fact 10000）を計算するもので，階乗を計算する関数factの定義はdoを用いた繰り返し的な典型的なものである．再帰的にしなかったのは，factの引数が大きい場合，スタックが足りなくなり，実行が途中で中断してしまうからである．関数factは，実行する前にコンパイルしたので，インタプリタ実行ではない．

extended_mulコンパイル時の最適化オプション（-O）なしで構築したKCLで（fact 10000）を実行した場合の実行時間を，表1に示す．10回計測し平均値をとったものである（以下同様）．この実行中565回のごみ集めが行われ，そのための時間も実行時間に含まれている．最適化オプションなしなので，gccがC言語の関数extended_mulを翻訳して生成したアセンブラコードは，（ロード・ストア命令movqを主とする不必要な命令を大量に含む）冗長なものである．mulx, adcx使用のバージョンは，オリジナルと比べて，約9%の実行時間短縮が得られている．

一方，extended_mulコンパイル時の最適化オプション（-O）ありで構築したKCLで（fact 10000）を実行した場合の実行時間を，表2に示す．この実行

表1：(fact 10000)の実行時間（-O最適化なし）

extended_mulのバージョン	実行時間	比
オリジナル	1.943	1.0
mulx, adcx使用	1.765	0.908

（単位は秒）

表2：(fact 10000)の実行時間（-O最適化あり）

extended_mulのバージョン	実行時間	比
オリジナル	1.744	1.0
mulx, adcx使用（手修正なし）	1.733	0.994
mulx, adcx使用（手修正あり）	1.728	0.991

（単位は秒）

中にも，565回のごみ集めが行われ，そのための時間も実行時間に含まれている．最適化オプションありなので，gccがC言語の関数extended_mulを翻訳して生成したアセンブラコードは，先述のような不必要な命令を含まず，簡潔なものである．“手修正”というのは，carryという変数を使ってC言語レベルでキャリーの受け渡しを行っていたものを，機械語レベルで行うよう，gccが生成したextended_mulのアセンブラコードを，手作業で修正したもので，機械語レベルで命令数が減る（図4参照）．コンパイラの最適化オプションありの場合は，オリジナルと比べて，手修正なしの場合は約0.5%の実行時間短縮が，手修正ありの場合は約0.9%の実行時間短縮が得られている．コンパイラの最適化オプションなしの場合に比べて，実行時間短縮（速度向上）の効果は，ごくわずかである．

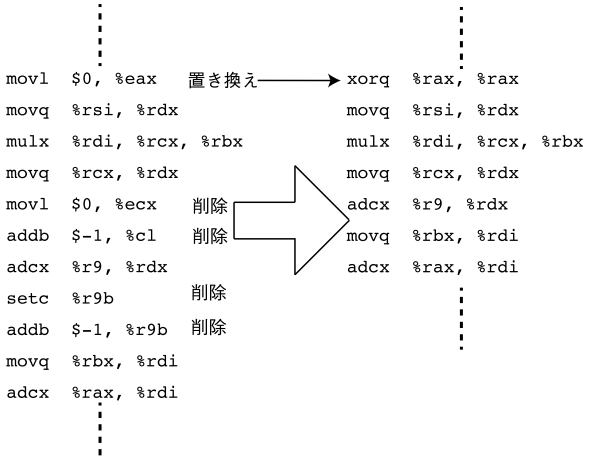

図4：gcc -Oが出力したアセンブラのコードの手修正

6 考察

gccの最適化オプションを指定しない場合，（fact 10000）の実行時間が約9%短くなったことは，mulxやadcxの使用の効果によるものだと推測される．実行時間には565回も行われたごみ集めなどextended_mul以外のものが含まれていることを考慮すれば，よく健闘していると判断される．

gccの最適化オプションを指定した場合は，実行時間の短縮がわずかであったのは，図2で示したオリジナルの4つの掛け算の全て（あるいはいくつかが）が同時に実行されたことが推測される．BroadwellはALUを複数持っているからである．最適化オプションを指定しないときは，冗長な命令が多かったため，掛け算の同時実行が行われなかったと推測される．

図4に示す手修正により得られたコードは，命令数が減ったことが，実行時間のわずかな短縮に繋がっている．C言語の変数によるキャリーの伝達のコストをみることができた．

7 まとめ

Broadwellの任意精度整数演算命令を，Lisp言語処理系のbignumの掛け算の中で用いることで，評価した．

mulx命令の意義は，64ビットの整数の積を確実に高速に行うことと，フラグを変更しないことであることが，明確にわかった．被乗数と乗数を32ビットずつに分割して4つの積を行う方法をとるとき，これらの積が同時実行できる場合は，mulx命令に実行効率が近づくが，生成される機械語命令の列によっては，常に同時実行できるとは限らないからである．

adcxがC言語から使えるようなったことは，C言語によるコード記述が簡潔になる利点があるが，C言語の変数を介するオーバーヘッドがあらわれる．コンパイラによる最適化においては，キャリーフラグのチェーンをみつけ，C言語の変数へのアクセスを排除して，adcx（やadox）の特徴を生かしたコード生成が行われることが望まれる．

参考文献

[1] Intel Corporation: *New Instructions Supporting Large Integer Arithmetic on Intel Architecture Processors*, Document number: 327831-001 (2012).
[2] Intel Corporation: *Large Integer Squaring on Intel Architecture Processors*, Document number: 328569-001 (2013).
[3] Steele, G. L. Jr.: *Common Lisp the language*, Digital Press (1984).
[4] Yuasa, T. and Hagiya, M.: *Kyoto Common Lisp Report*, Teikoku Insatsu Publishing (1985).
[5] Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol. 13, No. 3, pp. 284–295 (1990).
[6] Intel Corporation: *User and Reference Guide for the Intel C++ Compiler 15.0* (2015).
[7] 安本太一：Common Lisp言語処理系の64ビット化，愛知教育大学研究報告 自然科学編，五十三輯，pp. 27–32 (2004).
[8] 安本太一：Common Lisp言語処理系による64ビット環境の評価，愛知教育大学研究報告 自然科学編，五十五輯，pp. 9–13 (2006).

（2015年9月24日受理）